

Computational Complexity Theory - Course Notes

CCT Team, second revision

December 12, 2006

4 Lecture 4, 8/9-2006

4.1 Bi-Immunity

Definition 4.1 Let \mathcal{C} denote a class of languages. A language L is \mathcal{C} -biimmune if

1. No infinite subset of L is in \mathcal{C} , and
2. No infinite subset of L is in the complement of \mathcal{C} .

4.2 Reductions

Definition 4.2 A log-space map $r : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a map which is computable in $O(\log n)$ space by a deterministic Turing Machine.

Definition 4.3 Given languages L_1 and L_2 . We say that L_1 is log-space reducible to L_2 written $L_1 \leq_{\log} L_2$ whenever there is a log-space map r so that $x \in L_1 \Leftrightarrow r(x) \in L_2$ for all x in $\{0, 1\}^*$.

The subscript \log is typically omitted whenever the context makes it clear that we are talking about log-space reductions.

Lemma 4.1 Let $r : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be computable by a deterministic TM M in time $S(n) \geq \log n$. Then M uses at most $O(n \cdot c^{S(n)})$ steps to compute $r(x)$ where $n = |x|$.

Proof. For each possible configuration of the Turing Machine M , the read/write head can be positioned in n different positions on the input tape. Since M uses at most $S(n) \geq \log n$ cells on its work tape and each cell can hold two different say c different letters in each cell we get a total of $n \cdot c^{S(n)}$ possible configurations. Since the machine is deterministic and it halts, no configuration will occur more than once in any execution of M on x (otherwise the machine would not stop).

□

A consequence of the above lemma is that the machine M produce at most $O(n \cdot c^{S(n)})$ characters of output (since at most one symbol can be output at each step).

Theorem 4.1 *If $r_1, r_2 : \{0, 1\}^* \rightarrow \{0, 1\}^*$ are computable using space $S_1(n) \geq \log n$ and space $S_2(n) \geq \log n$ respectively then $r_1 \circ r_2$ is computable using space $S_1(n) + S_2(n \cdot c^{O(S_1(n))})$.*

Proof.

Let Turing machines M_1 and M_2 calculate the reductions r_1 and r_2 in space $S_1(n) \geq \log n$ and space $S_2(n) \geq \log n$ respectively.

The naive idea of computing $r_1 \circ r_2$ is; for a given input x run M_1 on x producing an output string y and run M_2 on y . Running M_1 on x takes space $S_1(n)$ of workspace producing an order of $m = nc^{O(S_1(n))}$ characters for y . Using M_2 on y takes up workspace $S_2(m)$. This yields a total space consumption of $S_1(n) + nc^{O(S_1(n))} + S_2(nc^{O(S_1(n))})$ which is certainly not the overall space bound we are out for.

The problem is that the intermediate storage of y . The solution is not to store all of y on a tape. Instead we create a new tape on M_1 storing a variable called i . M_1 is reprogrammed so that it for a given x computes until character number i of y is created (this character is written y_i). Now as M_2 needs to read character i of the input y it performs runs of M_1 letting the tape on M_1 holding the value i .

Representing i on M_1 only takes space $O(\log n)$ (we can represent the number i using binary numbers) which is fine. Machine M_1 uses space $S_1(n)$. By lemma 4.1 M_1 may at most produce output of size $nc^{S_1(n)}$. Using M_2 on input of this size yields a total upper space bound of $S_1(n) + S_2(n \cdot c^{O(S_1(n))})$.

□

Corollary 4.1 *If $L_1 \leq_{\log} L_2$ and $L_2 \leq_{\log} L_3$ then $L_1 \leq_{\log} L_3$.*

Proof. Using the above theorem with $S_1(n) = S_2(n) = \log n$ gives a space use of $\log(n) + \log(n \cdot c^{O(\log n)}) = \log(n) + \log(n \cdot O(n)) = O(\log n)$.

□

4.3 Hardness and completeness

Definition 4.4 *For any class of languages \mathcal{C} we define language L to be*

- hard for \mathcal{C} if $L' \leq_{\log} L$ for all $L' \in \mathcal{C}$.
- complete for \mathcal{C} if $L \in \mathcal{C}$ and L is hard for \mathcal{C} .

Exercise 4.1 Show that for “nice” $\mathcal{C} \in \{L, NL, P, NP, PSPACE, \dots\}$ it holds that if $L_1 \in \mathcal{C}$ and $L_2 \leq_{\log} L_1$ then also $L_2 \in \mathcal{C}$.

Theorem 4.2 If $\mathcal{C}_1 \subseteq \mathcal{C}_2$, \mathcal{C}_1 “nice” and L is complete for \mathcal{C}_2 then $\mathcal{C}_1 = \mathcal{C}_2 \Leftrightarrow L \in \mathcal{C}_1$.

We are interested in the nice class NF since the above theorem gives rise to nice applications.

Theorem 4.3 (Papadimitriou 16.2) The *GAP* problem is *NL*-complete.

Proof.

Given a graph $G = (V, E)$ with $V = \{1, \dots, n\}$ and $E \subseteq V \times V$. *GAP* is the decision problem: Is there a path from vertex 1 to vertex n ? Formally $\text{GAP} = \{\langle G, i, j \rangle \mid \text{There is a path from } i \text{ to } j \text{ in } G\}$

$\text{GAP} \in \text{NL}$:

Using a nondeterministic TM we can solve the problem using the following algorithm:

```

i := 1
for (c := 1 to n) do
  Guess j;
  if (i, j) not in E, then return Reject
  else
    i := j
    if i = n then return Accept
return Reject

```

Since the length of a possible path is at most n long we will only allow the algorithm to run in at most n steps. This is handled by the `for`-loop. If n is found we accept otherwise we return reject. The representation of the variables i, j, c are binary so we do have that $\text{GAP} \in \text{NL}$.

$\forall : L \in \text{NL} : L \leq_{\log} \text{GAP}$: Given a language L decided by the NL -machine M . Since the machine is a $\overline{\text{NL}}$ -machine it only uses logarithmic space during its computation. The idea is to create a graph where nodes are Turing machine configurations and edges are Turing machine transitions between vertices's. We will construct the reduction r so that:

x is accepted by M iff
 $r(x)$ has a path from node 1 to some “accepting” node say k

We know that there are $O(nc^{\log n})$ possible configurations of such a machine. We let C denote the number of such configurations. We will now imagine that we have an enumeration of the possible configurations. By the following algorithm we create an adjacency graph mentioned before:

```
for i := 1 to C do
  for j := 1 to C do
    If machine M runned on i yields output j
    then G[i, j] := 1

    If machine M runned on i do not yield output j
    then G[i, j] := 0
```

Now we can use the algorithm for deciding **GAP** on the graph G simulating the behavior of M . It is true that the above algorithm runs in log-space (think about this).

□

Now we can understand the complexity class **NL** in terms of **GAP** and vice versa.

Theorem 4.4 *It holds that $\text{NL} \subseteq \text{P}$.*

Proof. It is enough to show that $\text{GAP} \in \text{P}$, since **GAP** is **NL**-complete and **P** is closed under reductions.

To see that $\text{GAP} \in \text{P}$ consider the following algorithm

- 1 Input $\langle G, s, t \rangle$
- 2 Mark node s
- 3 Repeat
- 4 for each node b of G
- 5 if there is an edge (a, b) where a is marked
- 6 then mark b .
- 7 until no new nodes are marked
- 8 if t is marked accept, else reject.

Theorem 4.5 (Savitch's theorem) $\text{NL} \subseteq \text{DSPACE}(O(\log^2 n))$.

Proof. It is enough to show that $\text{GAP} \in \text{DSPACE}(O(\log^2 n))$. Given an instance $\langle G, s, t \rangle$ of GAP. We will write $\text{PATH}(u, v, l)$ if there is a path from u to v in G of length at most l . Now we first discover that

$$\text{PATH}(u, v, l) \leftrightarrow \exists a : \text{PATH}\left(u, a, \frac{l}{2}\right) \wedge \text{PATH}\left(a, v, \frac{l}{2}\right)$$

Meaning that if there is a path from u to v of length at most l then there is an intermediate vertex a on the halfway on that path. Using this observation we can formulate the following recursive procedure calculating $\text{PATH}(a, b, l)$:

```

1  boolean PATH(a, b, l)
2  If l = 0 then FALSE
3  If l = 1 and b ∈ Adj(a) then return TRUE
4  Else
5  for every vertex v in G
6    if PATH(a, v, ⌈l/2⌉) and PATH(v, b, ⌊l/2⌋)
7    then return TRUE
8  otherwise return FALSE

```

The above function evaluates to true whenever there is a path from a to b in graph G of length at most l . Letting $n = |G|$ it is clear that any path from s to t in G is at most of length n . Hence we can determine whether $\langle G, s, t \rangle$ is a GAP instance by checking that $\text{PATH}(s, t, n)$.

The space used to compute $\text{PATH}(s, t, n)$ can be found the following way. The storage of variables a, b, l and v requires space $4 \cdot O(\log n)$ i.e $O(\log n)$. The algorithm can only perform at most $2 \cdot \log n$ function calls since n is half-ed in every function call. Before each function call is performed the current stack frame is stored, this takes $O(\log n)$ space. The overall space consumption is hence $2 \log n \cdot O(\log n)$ which is $O(\log^2 n)$ as claimed.

4.4 Upwards translation

Theorem 4.6 (Generalized version of Savitch theorem) . *Let $S(n) \geq \log n$ be a space constructible bound. Then $\text{NSPACE}(S(n)) \subseteq \text{DSPACE}((S(n))^2)$.*

Proof. Let $L \in \text{NSPACE}(S(n))$ be decided by a non-deterministic Turing machine M_L . Define the padded language $L' := \{ \langle x, 1^{\exp(S(|x|))} \rangle \mid x \in L \}$. The language L' can be decided by a non-deterministic Turing machine using logarithmic space i.e. $L' \in \text{NL}$. This is true since: for a given $y = \langle x, 1^k \rangle \in L'$ check that k is equal to $\exp(S(|x|))$ by first counting the number of one's using binary

representation. Having done that check that the number is an exponentiation of 2 by repeated shift right operations. If this is not the case reject. Otherwise simulate x on M_L and reject if M_L rejects, and accept x if M_L accepts.

By theorem 4.5 we have that $L' \in \text{DSPACE}(\log^2 n)$. This means that there is a deterministic Turing machine M which decides L' in space $\log^2 n$.

Now L can be decided by the following deterministic machine:

- On input x
1. Calculate $y = \langle x, 1^{\exp(S(|x|))} \rangle$
 2. Run M on y .
 3. If M accepts *accept* otherwise *reject*.

The space consumption of the above machine is $n + \underline{\exp(S(n))} + \log^2(n + \exp(S(n)))$, $n = |x|$ which is far too much. The problem is the underlined part of the above sum.

We can avoid this space consumption by combining lines 1 and 2 in the above algorithm and use the same trick as in theorem 4.1. Now the total space usage is bounded by $O(\log^2(n + \exp(S(n))))$ which is equal to $O(\log^2(\exp(S(n)))) = O(S(n)^2)$. We hereby conclude that $L \in \text{DSPACE}((S(n))^2)$.

Corollary 4.2 PSPACE = NPSPACE

4.5 Co-classes

Definition 4.5 Define the complement coC of a class \mathcal{C} as $\{\{0, 1\}^* - L \mid L \in \mathcal{C}\}$.

- If $P \neq NP$ then SAT, IP, CLIQUE $\notin P$.
- If $NP \neq \text{coNP}$ then FACTORING is not NP-complete.